L10 - More arrays, Library classes

February 11, 2020

1 More Arrays

1.1 2-dimensional array

Java's `Array` syntax supports multiple dimensions. A 2d array may be used as a game board, a matrix, etc. There is no limit to the number of dimensions.

The multidimensional array can be treated as an array *of arrays*.

1.1.1 Array of array iteration

It is possible to treat the 2d array more directly as an "array of arrays":

[ ]:
```
for ( int row = 0; cells < cells.length; row++ )
{
    Cell[] nextRow = cells[row]
    for ( int col = 0; col < cells.length; col++ )
    {
        // do something
    }
}
```

1.2 The Brain Project

The *Brian's Brain* project is a 2d cellular automaton with some graphics of some sort.

The *Moore Neighbourhood* of any given cell is the set of cells surrounding it, including the corners. These are the cells whose state will affect the next state of the centre cell.

- from wikipedia: a **Moore machine** is a finite-state machine whose output values are determined only by its current state.

The rules for updating a cell's status are:

- a cell that is *alive* changes to *dying*
- a cell that is *dying* changes to *dead*
- a cell that is *dead* changes to *alive* if exactly two of its neighbours are alive; otherwise it stays dead

Lab 5: modify this thing to play Conway's Game of Life instead

[3]:
```java
import java.util.*;

/**
 * A cell in a 2D cellular automaton.
 * The cell has multiple possible states.
 * This is an implementation of the rules for Brian's Brain.
 * @see https://en.wikipedia.org/wiki/Brian%27s_Brain
 *
 * @author David J. Barnes and Michael Kölling
 * @version  2016.02.29
 */
public class Cell
{
    // The possible states.
    public static final int ALIVE = 0, DEAD = 1, DYING = 2;
    // The number of possible states.
    public static final int NUM_STATES = 3;

    // The cell's state.
    private int state;
    // The cell's neighbors.
    private Cell[] neighbors;

    /**
     * Set the initial state to be DEAD.
     */
    public Cell()
    {
        this(DEAD);
    }

    /**
     * Set the initial state.
     * @param initialState The initial state
     */
    public Cell(int initialState)
    {
        state = initialState;
        neighbors = new Cell[0];
    }

    /**
     * Determine this cell's next state, based on the
     * state of its neighbors.
     * This is an implementation of the rules for Brian's Brain.
     * @return The next state.
     */
```

```
    public int getNextState()
    {
        if(state == DEAD) {
            // Count the number of neighbors that are alive.
            int aliveCount = 0;
            for(Cell n : neighbors) {
                if(n.getState() == ALIVE) {
                    aliveCount++;
                }
            }
            return aliveCount == 2 ? ALIVE : DEAD;
        }
        else if(state == DYING) {
            return DEAD;
        }
        else {
            return DYING;
        }
    }

    /**
     * Receive the list of neighboring cells and take
     * a copy.
     * @param neighborList Neighboring cells.
     */
    public void setNeighbors(ArrayList<Cell> neighborList)
    {
        neighbors = new Cell[neighborList.size()];
        neighborList.toArray(neighbors);
    }

    /**
     * Get the state of this cell.
     * @return The state.
     */
    public int getState()
    {
        return state;
    }

    /**
     * Set the state of this cell.
     * @param The state.
     */
    public void setState(int state)
    {
        this.state = state;
```

```
    }

}
```

[5]:
```
int numRows = 5;
int numCols = 5;

Cell[][] cells; // a 2d cell array

cells = new Cell[numRows][numCols]; // instantiate the top-level cellular␣
 ↪automaton

for ( int row = 0; row < numRows; row++ )
{
    for (int col = 0; col < numCols; col++ )
    {
        cells [row][col] = new Cell();
    }
}
```

1.2.1 getNextState()

This method implements the state machine for each cell. Note that the final else only executes if the cell is currently *alive*.

1.2.2 setNeighbours()

An `ArrayList` of cells that includes all the cell's neighbours. These neighbours are set up by the `setupNeighbors()` method in the `Environment` class that initializes the grid. The grid has a toroidal arrangement (like a Karnaugh map).

1.3 Library Classes

The Java class library has thousands of classes with tens of thousands of methods; many are useful and make life easier. They are arranged into *packages*.

A competent Java programmer must be able to work with the the libraries. You should know some important classes by name and be able to find out about other classes.

Relax: we only need to know about the *interface*, not the *implementation*.

1.3.1 A digression

13 students in class rn; new record.

1.4 Tech Support Project

1.4.1 Main loop structure

```
boolean finished = false;
```

```
while !finished
{
    do something

    if ( exit condition )
    {
        finished = true;
    }
    else
    {
        do something more
    }
}
```

This is a common iteration pattern.

The project consists of three classes; `SupportSystem` is the overall system; `Responder` responds to queries; and `Reader` reads text input.

1.4.2 Reading class docs

Class docs are in HTML and can be read in a web browser. They describe the *class API*: the *application programming interface*; these are the interface descriptions on all library classes.

Hit `Ctrl-J` in BlueJ to read the JavaDoc. If you write your comments in the correct format (which is suggested by IntelliCode in VS Code), they will automatically get read into the HTML file by JavaDoc, which is kinda cool.

1.4.3 A further digression

Down another one; 12 students now.

1.4.4 Interface vs Implementation

The documentation includes everything you need to know in order to used it: its constructors and methods.

It does *not* include the class's private fields or internal workings. These are presumably subject to change, but so long as the class maintains its API "contract" with the outside world we don't need to concern ourselves.

For example, for `Shoelace.startsWith(String sub)`:

- return type: `boolean`
- returns `true` if `sub` is the first few letters of `Shoelace`

Mildly interesting note: `String` is an immutable object; if you want to change one, you need to return some other modified version then assign it in overtop.

1.4.5 Using Library Class

Classes are organized into packages. Basic classes like `String` are already included in `javalang` so don't need to be imported.

Single classes can be imported: `import java.util.ArrayList`

Or loads at once: `import java.util.*`

1.4.6 Randomness

[11]:
```
// generate a pseudorandom number

import java.util.Random;

Random rand = new Random();

int num = rand.nextInt();
System.out.println(num);
```

```
-2102718399
```

1.4.7 Parameterized Classes

The documentation provides for a type parameter. These appear in the parameters and return types of the methods. Types used in the docs can be placeholders.

Parmeterized classes are sometimes knowns as *generic types*.

1.5 Set

A collection without duplicates.

[16]:
```
import java.util.HashSet;

HashSet<String> mySet = new HashSet<>();

mySet.add("one");
mySet.add("two");
mySet.add("three");

System.out.println("First Print:");

for ( String element: mySet )
{
    System.out.println(element);
}

mySet.add("two");
mySet.add("seven");
```

```
System.out.println();

System.out.println("Second Print:");

for ( String element: mySet )
{
    System.out.println(element);
}
```

```
First Print:
one
two
three

Second Print:
one
seven
two
three
```

1.6 Map

Creates associations: contains *pairs* of objects, and parameterized with two types. Each pair consists of a key and a value; and lookup works by supplying a key and retreiving a value.

1.7 Tokenizing Strings

From tech-support-complete:

[18]:
```
/**
     * Read a line of text from standard input (the text terminal),
     * and return it as a set of words.
     *
     * @return  A set of Strings, where each String is one of the
     *          words typed by the user
     */
    public HashSet<String> getInput()
    {
        System.out.print("> ");                // print prompt
        String inputLine = reader.nextLine().trim().toLowerCase();

        String[] wordArray = inputLine.split(" ");  // split at spaces

        // add words from array into hashset
        HashSet<String> words = new HashSet<>();
        for(String word : wordArray) {
            words.add(word);
        }
```

```
        return words;
    }
```

[ ]: